Cours de compilation
3ème partie
Analyse lexicale

Jacques Ferber

1. Introduction

L'analyse lexical consiste à segmenter un texte source en un ensemble de mots que l'on appelle traditionnellement «tokens» (leur terme exact est «lexème», ce qui signifie unité lexicale). Il s'agit d'une part de déterminer la suite des caractères comprenant le token, et d'autre part d'identifier le type de token (identificateur, nombre, opérateur, etc.).

Par exemple le texte suivant:

```
toto1:=t25+2.5*toto1;
```

doit être capable de comprendre qu'il s'agit de la suite de tokens suivants:

ident	toto1
affectation	:=
ident	t25
op.arith	+
flottant	2.5
op.arith	*
ident	toto1
séparateur	;

Comment faire? Nous envisagerons d'abord des techniques simples dans lesquelles l'analyseur lexicale sera construit « à la main ». Puis, nous verrons que cette analyse peut être formalisée par l'intermédiaire d'automates finis. Enfin, nous essayerons d'en dégager une technique générale pour construire ce que l'on appelle des générateurs d'analyseurs (ex : le programme *lex* sur Unix).

2. Approche naïve

L'approche naïve consiste à traiter le problème directement en essayant d'écrire directement un programme à partir d'une idée intuitive de ce qu'est un mot.

On supppose que l'on dispose des fonctions de manipulation de table des symboles suivantes:

```
est_dans_table(<chaine>) // regarde si une chaine est dans table
                         // retourne le couple (numero, type) si oui
```

```
                    // retourne 0 si non (la table commence a 1)
ajouter_token(<chaine>,<entier>)    // ajoute un token a la table
        // retourne un couple
```

Voici une procédure de lecture de quelques informations:

```
Lecture_identificateur() {
   Tant que((c=lireCar()!=EOF) && type(c)== CHAR_ALPHA)
      alors mettre c dans Buffer
   si ((r = est_dans_table(Buffer)) == 0) alors
      retourner(ajouter_token(buffer,TYPE_IDENT))
   sinon
      retourner(r)

lecture_entier() {
   Tant que((c=lireCar()!=EOF) && type(c)== CHAR_INTEGER)
      alors mettre c dans Buffer
   si ((r = est_dans_table(Buffer)) == 0) alors
      retourner(ajouter_token(buffer,TYPE_INTEGER))
   sinon
      retourner(r)
```

Quelques problèmes avec cette approche:

1. Il faut penser à remettre les caractères qui ont été lus en trop dans le flot d'entrée. Ceci est un tout petit pb technique est facilement résolu. Ce problème est facilement résolu soit par la lecture d'un caractère en avance, soit par l'utilisation d'une fonction telle que `ungetc()` en C qui replace un caractère dans le flux de sortie.
2. Plus gênant: difficile de prendre en compte des lexèmes constitués de plusieurs types de caractères. Par exemple, un identificateur est composé d'un caractère alphabétique suivi par une suite quelconque de caractères alphabétique ou numériques.
3. Comment déterminer quelle type de procédure appeler?
4. Suppose que l'on écrive tout à la main. La moindre modification nécessite une réécriture complète de l'ensemble.

Pour résoudre ces problèmes (sauf le 4) on utilise des diagrammes de transitions. Pour généraliser l'approche et réaliser automatiquement des analyseurs lexicaux on utilise la technique des expressions régulières.

3. Automates

3.1 Introduction

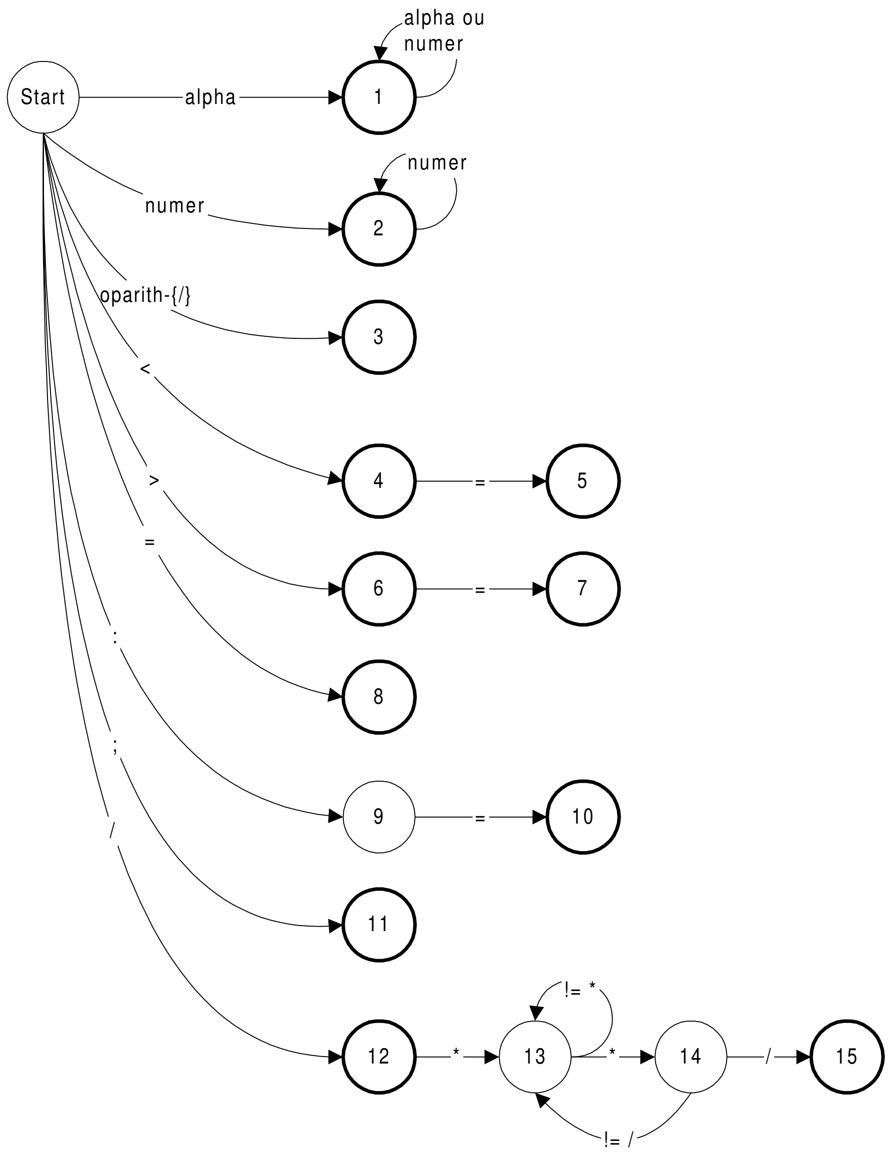
Idée: on considère que lire un caractère peut faire passer d'un état dans un autre. Par exemple, lire un opérateur '<' fait passer dans l'état de lecture des opérateurs arithmétiques, si on lit ensuite un '=' on reste dans cet état (et on a terminé de lire un opérateur arithmétique) sinon on sait que l'on a terminé

Définition : un automate est défini sous la forme <S, δ, A, s0, F> où S est un ensemble d'états, A est un alphabet (un ensemble de caractères), s0 est l'état initial de l'automate, F $\subseteq$ S est l'ensemble des états finaux de l'automate, et δ est la fonction de transition de l'automate, $\delta : A \times S \rightarrow S$.

Les automates sont souvent donnés sous la forme d'un graphe: les états sont les noeuds du graphe et les arcs correspondent à la fonction de transition.

3.2 Analyseur lexical et automate

L'automate correspondant à un sous-ensemble de Pascal (dans lequel on suppose que les mots clés sont directement placés dans la table des symboles) est le suivant (les états finaux sont présentés en gras):



Automate d'un analyseur lexical

Il faut maintenant attacher des actions à chacun des états de l'automate et implémenter cet automate.

Pour implémenter un automate on considère

```
buffer=&Buffer[0];
while(1) {
   c = lireChar();
   switch(etat){
      case: 0
         switch(type(c)){
              buffer*=c;buffer++;
              case ALPHA: etat=1;break;
              case NUMER: etat=2;break;
              case OPARITH-{/}: etat=3; break;
              case '<': etat=4; break;
              case '>': etat=6; break;
              case '=': etat=8; break;
              case ':': etat=9; break;
              case ';': etat=11; break;
              case '/': etat=12; break;
              default: erreur;
         }
         break;
      case 1:
         if (type(c) == ALPHA ou type(c) == NUMER) {
              buffer*=c;buffer++;
              etat=1;
         }
         else {
              buffer*='\0'; remettreCar();
              return(ajouter_token(Buffer,TYPE_IDENT));
         }
         break;
      case 2:
         if (type(c) == NUMER){
              buffer*=c;buffer++;
              etat=2;
         }
         else {
              buffer*='\0';remettreCar();
              return(ajouter_token(Buffer,TYPE_NUMER));
         }
         break;
      case 3:
         return(ajouter_token(Buffer,TYPE_OPER)); break;
      case 4:
         if (c == '=') {
              buffer*=c;buffer++;
              etat = 5;
         } else {
              remettreCar();
              return(est-dans("<"));
         }
```

```
            break;
{{à finir!!}}
        }
```

3.3 Expressions régulières

3.3.1 Définition

Une expression régulière est une notation pour décrire un langage régulier.

Soit A un alphabet (un ensemble de lettres), une expression régulière est don

1. Les éléments de A, ε et ∅ sont des expressions régulières.
2. Si α et β sont des expressions régulières, alors (α | β), (αβ) et α* sont des expressions régulières. (α | β) représente l'union, (αβ) la concaténation et α* la répétition (un nombre quelconque de fois). ε est l'élément neutre par rapport à la concaténation et ∅ est l'ensemble vide de caractère, neutre par rapport à l'union.

Exemple : (a|b)* abb,

Identificateur = alpha (alpha | numer)*

En fait, les expressions régulières sont beaucoup plus puissante que ce dont on a besoin lorsqu'on fait de l'analyse lexicale.

3.3.2 Expressions régulières et automates

Les expressions régulières sont intéressantes parce qu'on peut définir, à partir de ces expressions, des automates qui reconnaissent si une expression d'un langage peut être décrite par un langage particulier. Et il existe des algorithmes permettant de passer d'une expression régulière à un automate.

Malheureusement, il est difficile d'écrire simplement les automates qui correspondent à une expression régulière un peu compliquée (comme celle qui est donnée en exemple). Il faut pour cela passer par le concept d'automate non-déterministe.

Un automate non-déterministe est un automate qui présente la particularité suivante :

1. Plusieurs arcs reconnaissant le même caractère peuvent « sortir » du même état.
2. Il peut y avoir des transitions, notées ε, qui ne correspondent à une aucune reconnaissance de caractères.

Exemple d'automate non-déterministe :